electronics

MDPI

Article

Optical Recognition of Handwritten Logic Formulas Using Neural Networks

Vaios Ampelakiotis [1], Isidoros Perikos [1], Ioannis Hatzilygeroudis [1,*] and George Tsihrintzis [2]

[1] Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece; ampelakiot@ceid.upatras.gr (V.A.); perikos@ceid.upatras.gr (I.P.)
[2] Department of Informatics, University of Piraeus, 18534 Piraeus, Greece; geoatsi@unipi.gr
* Correspondence: ihatz@ceid.upatras.gr

Abstract: In this paper, we present a handwritten character recognition (HCR) system that aims to recognize first-order logic handwritten formulas and create editable text files of the recognized formulas. Dense feedforward neural networks (NNs) are utilized, and their performance is examined under various training conditions and methods. More specifically, after three training algorithms (backpropagation, resilient propagation and stochastic gradient descent) had been tested, we created and trained an NN with the stochastic gradient descent algorithm, optimized by the Adam update rule, which was proved to be the best, using a trainset of 16,750 handwritten image samples of 28×28 each and a testset of 7947 samples. The final accuracy achieved is 90.13%. The general methodology followed consists of two stages: the image processing and the NN design and training. Finally, an application has been created that implements the methodology and automatically recognizes handwritten logic formulas. An interesting feature of the application is that it allows for creating new, user-oriented training sets and parameter settings, and thus new NN models.

Keywords: optical character recognition; logic formulas; neural networks; resilient propagation; OpenCV; Encog

Citation: Ampelakiotis, V.; Perikos, I.; Hatzilygeroudis, I.; Tsihrintzis, G. Optical Recognition of Handwritten Logic Formulas Using Neural Networks. *Electronics* **2021**, *10*, 2761. https://doi.org/10.3390/electronics10222761

Academic Editors: Juan M. Corchado, Stefanos Kollias and Javid Taheri

Received: 18 October 2021
Accepted: 9 November 2021
Published: 12 November 2021

1. Introduction

Optical character recognition (OCR) aims to formulate methods that deal with the efficient recognition of written text. Initially, OCR was used for the identification of postal codes, license plates, bank checks, and later, for converting images of text into editable digital files. These days, it assists in more and more human tasks such as automated validation of private documents (passports, identities, official forms), making images of printed text searchable by engines, helping blind people recognize text by reading it, extracting information from image text and adding to calendars and contacts.

Handwritten character recognition (HCR) is the most challenging subfield of the OCR field (the other being typed character recognition—TCR), given that many, sometimes diverse, case-based parameters should be considered [1]; these include the surrounding context of the target text (e.g., text in nature scenes, text within shapes or drawings), the hugely variant handwriting styles, the way that characters are written (e.g., different pen stroke widths, interconnected and skewed characters), the quality of the image (e.g., poor resolution, bad lighting conditions) [2,3]. Even if we assume all those challenges are solved, there can be cases of isolated characters that can be misidentified even by humans, although we are able to identify letters in formed words based on context.

Many of the approaches that tackle the HCR problem use some type of artificial neural network (NN) [4]. The most common architecture of NN, called feedforward multilayer network, consists of neurons that are connected to each other and are organized in layers: input layer (every neuron receives a value from an input vector), output layer (every neuron produces a final output) and hidden layers (transfer output of the previous layer to the next one). Each neuron, to propagate its input to its output (or the next neuron's

input), has a filter defined by an activation function, which is usually non-linear. An NN is trained via a training data set, which we call 'trainset', to specify a character recognition model. In our case, the size of the input layer is defined by the resolution size of each character, whereas the output layer's size by the total characters to be recognized. Initially, the number of hidden layers is chosen arbitrarily and afterward is updated by trial and error. During the training process, every connection has a weight that is changed according to the calculation of an error function. Performance of the NN is tested via a test data set, which we call 'testset'.

In recent years, quite sophisticated deep learning NNs, such as convolution neural networks (CNNs), have been used for HCR [5,6]. However, it may still be challenging to use simpler networks that achieve comparable or adequate accuracy levels for the problem at hand. Our problem is that of converting handwritten logical formulas into typed ones for educational purposes in offline mode.

In this work, we use a dense multilayer perceptron (MLP), in which neurons of a layer are fully connected with neurons from the previous and next layer and the connections have a forward direction. We use and compare three supervised gradient descent training algorithms: backpropagation (standard version), resilient propagation—RPROP (an improvement of backpropagation where weight changes are affected only by the gradient's sign), and stochastic gradient descent—SGD (where, in each epoch, a random portion of the training set is chosen to calculate the average error function and weights are changed via the Adam update rule) [7].

We consider HCR as a multi-class classification problem. To this end, we defined an image preprocessing methodology for dataset creation from text images to train and test the NN. Created datasets map input pixel-based handwritten character representations to output one-hot encodings of typed characters. The pixel values belong to {0, 1}. We experimented with several NN structures and activation functions. We also experimented with different hyperparameters' values. The hyperparameters considered were learning rate, momentum, batch size and the number of epochs.

Furthermore, we developed a Java application that implements the whole process: takes as input an image of handwritten logic formulas, preprocesses it, specifies areas of handwritten logic formulas, recognizes every character with a pre-trained NN, and prints the logic formulas in an editable text file. In addition, the application provides the capability of creating a new training set and specifying a new NN model (i.e., configure and train the NN from scratch).

Therefore, the contributions of this work are as follows:

- Use of NN for HCR in a new problem domain (handwritten logical formulas), where special characters, apart from English letters and digits, are also used.
- Introduction of an image preprocessing methodology, especially as far as the "Line detection" subprocess is concerned, for dataset creation.
- A thorough study of the effects of the changes in the values of the training hyperparameters on the performance of the NN.
- Development of a software tool that implements the introduced methodology and allows re-creation of the dataset and re-configuration of the NN model.

The rest of the paper is structured as follows. Section 2 presents related works on HCR, mainly for the English language. Section 3 presents the proposed methodology, describes its stages and analyzes its functionality. Section 4 presents the formulation of a novel dataset of handwritten logic formulas, the experimental study and the collected results. Section 5 presents the developed application that implements the methodology and is used to recognize handwritten logic formulas in real conditions. Section 6 discusses the main findings of the experimental study. Section 7 presents a comparison of our approach with other similar approaches. Finally, Section 8 concludes the paper and provides directions for future work.

2. Related Work

In recent years, many approaches have been designed and applied to accomplish the HCR task. In this section, we present recent works mainly concerning HCR for handwritten or typed English language, based on NNs.

Perwej and Chaturvedi [8] propose the use of a feedforward MLP network for recognizing handwritten lowercase English alphabet characters. They created a dataset of 650 samples (25 for each character) from different writing styles and a grid size of 5×5 binary pixels. The network had 25 neurons at the input layer, 2 hidden layers of 25 neurons each (with logistic sigmoid and tanh activation functions, respectively), and a 0.05 learning rate. The final accuracy achieved was 82.5%, and the error margin was due to weakness of distinction between look-alike characters (e.g., c–e, i–j, u–v, etc.)

An interesting approach to HCR is presented in the work of Kader and Kaushik [9], which focuses on the size and color-invariant digits and capital English letters of "Times New Roman" font using a feedforward NN without any hidden layers. The whole recognition process is divided into four basic steps: preprocessing (digitization, noise removal and boundary detection), normalization (12×8 grid size of binary pixels), network establishment (input layer of 96 neurons and output layer of 36 neurons) and recognition. The trainset size was around 5–10 samples per character, and the testset size was more than 20 samples per character. After training with different subsets of the trainset, the accuracies achieved were 99.99% for digits, 98% for letters and about 94% for both mixed. The interesting conclusion is that the proposed system produced excellent results for numeric digits and letters when trained and tested separately, and a little lower when they were trained together.

Choudhary et al. [10] underline that the main factors judging HCR's accuracy and capability are the choice of the classifier and the approach used to extract character features. They focus on the recognition of handwritten English alphabet characters by creating a dataset of 1300 samples from 10 people with a grid size of 15×12 binary pixels. In a MATLAB environment, they preprocessed every sample by thresholding it with a global value, followed by resizing, cropping and denoising. A feedforward NN with an input layer of 180 neurons, one hidden layer of 80 neurons and output layer of 26 neurons with one-hot encoding was trained using a backpropagation algorithm and adaptive learning rate. Furthermore, the activation function of the hidden and output layer was tanh. The average overall recognition accuracy achieved was 85.62%.

Katiyar and Mehfuz [11] use an NN of 144-100-90-6 architecture, where they make use of 144 hybrid features, extracted by four different methods (boxing, diagonal distance, mean pixel value, image gradient). Backpropagation gradient descent with momentum and adaptive learning is used for training. The CEDAR (Centre of Excellence for Document Analysis and Recognition) benchmark dataset of the English alphabet is used for training and evaluation (19,145 isolated characters for training and 2183 for testing). A recognition rate of 93.23% was achieved.

The work in [12] concerns the recognition of 62 typed English characters (alphabetic and digits). A feedforward NN with one hidden layer is used, implemented in MATLAB. A total of 558 samples of 62 typed characters each, where each character was in nine different fonts, were used for feature extraction and training. The backpropagation algorithm was used. The final architecture was 96-79-62 and achieved a recognition rate of 93%.

In [13], a methodology was developed to convert handwritten text, including only alphabetic English characters, into typed text, using a multilayer feedforward NN, without using feature extraction. The methodology includes image acquisition, preprocessing, segmentation, classification, and recognition. After recognition, in a postprocessing stage, the recognized characters are printed in a structured format. The used dataset consists of 4840 characters (4400 for training and 440 for testing), produced from text written by 100 different people. Handwritten characters were resized into 30×20 pixels, so each resized character consists of 600 pixels, considered as features. The final NN had two hidden layers of 100 neurons with a logistic sigmoid activation function. The proposed

system achieved an accuracy of up to 90.19%. A system implementing the methodology is also reported.

The work of Chen [14] uses two smaller subsets of MNIST, one that had only digits 0 and 1 (12,665 samples for the trainset and 2115 for the testset) and the other only digits 3 and 5 (11,552 samples for the trainset and 1902 for the testset) and trained two feedforward NNs as a binary logistic regression model with the SGD training algorithm and batch size of 1. Furthermore, he conducted experiments by varying the trainset size from 5% to 100%. The results showed that smaller sizes of trainset led to insignificant, lower accuracies compared to the highest accuracies achieved with full-size. More specifically, the accuracy range was 97–99% for the 0–1 pair and 85–94% for the 3–5 pair.

Jana et al. [15] present a comparative study of multilayer perceptron (MLP), recurrent neural network (RNN) and convolutional neural network (CNN), of which CNN provided the highest accuracy. They focus on the recognition of handwritten digits using the famous MNIST dataset. The MNIST dataset has a trainset of 60,000 and a testset of 10,000 digits samples, normalized in 28 $\times$ 28 pixel size. After training CNN with two methods, they achieved 98.92% accuracy with method 1 and 98.85% with method 2. The latter method was finally proposed because it needed less training time.

Yousaf et al. [16] present a methodology for handwritten English character (alphabetic and digital) preprocessing and recognition through an NN with one hidden layer, without using feature extraction. Preprocessing consists of binarization (by Otsu's global thresholding method), segmentation, filtering, and dynamic resizing. They used the HCD dataset, which consists of 19,422 alphabets and 7720 digits, produced from text written by 150 different writers. Handwritten characters are resized into 60 $\times$ 40 pixels, so each resized character consists of 2400 pixels, considered as features. Two NNs were configured with the best results, one for alphabet recognition with 240 hidden neurons and 96.98% precision, and the other for digit recognition with 120 hidden neurons and 98.08% precision.

Kosykh et al. [17] discuss the task of optimizing parameters of NNs for image recognition using MATLAB and Hadoop tools. They used the MNIST digit dataset to evaluate the accuracy of the network, dependent on choosing the number of hidden layer neurons, choosing the optimal training algorithm, and using parallelization with the MATLAB Distributed Computing Server. By testing a range of (10,300) neurons in a hidden layer, they concluded that (considering 784 input layers and 10 output layer sizes) the best accuracy (95.7%) was achieved with 145 neurons. As for the training algorithms, they tested RPROP, SGD, the Fletcher–Powell conjugate gradient, the Powell–Bill method, the Polac–Ryber method and the one-step algorithm (cutting planes method). Among them, RPROP turned out to be the fastest with lower accuracy, but on those terms, SGD was the optimal choice (95.7% accuracy). Finally, they show that multiple CPU cores on distributed computing can speed up calculations and training time.

Parthiban et al. [18] use a recurrent neural network (RNN) for handwritten English character recognition. The IAM dataset is used, which includes images of 79 distinct characters, resized in 32 $\times$ 80 pixels. They achieved 90% accuracy.

Recently, combinations of CNNs with other classifiers have been used for HCR. For example, in [19], combinations of various CNN types (AlexNet, ZfNet, LeNet) with the error-correcting output code (ECOC) are examined. The combination with AlexNet achieves the best accuracy (97.71%). The NIST handwritten character image dataset was used for training and validation. Similarly, in [20], CNN and SVM are combined. Using the handwritten digit images of the MNIST dataset, an accuracy of 99.28% is achieved.

3. Methodology

In this section, we present the methodology followed to create a model for character recognition and analyze its stages. The methodology has two main stages:

- Image preprocessing,
- Neural network design and training.

3.1. Image Preprocessing

Image preprocessing is the first stage in creating a model for handwritten text recognition and aims at converting images in a form that represents (some of) their features to be able to produce a suitable dataset for neural network training. In Figure 1, the main steps of our image preprocessing stage are depicted. Each step may include other sub-steps, which are not displayed for simplicity reasons. *Creation of handwritten text images* is the step where specific text of logic formulas handwritten by different people on paper are collected and scanned to obtain corresponding text images. More specific information is given in Section 4.1.

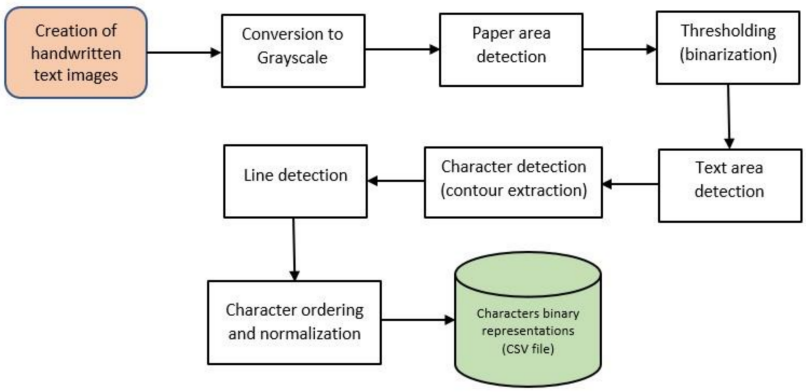


Figure 1. Image processing flowchart.

3.1.1. Conversion to Grayscale

In this step, considering that our application should have the ability to detect both scanned and camera images, a black 20-pixel wide *padding* is initially inserted to help detection of the paper area. In the case of camera images, this demands that the background must be plain black. Afterward, images are converted to grayscale with pixel values from 0 to 255.

3.1.2. Detection of the Paper Area

The aim of this step is to find the largest rectangle shape in an image, which will be the paper area. First, we *smooth* the image with a bilateral filter to preserve edges and then *blur* the white area. Next, we apply a global *threshold* with the Otsu method [21], detect the paper's largest contour, and apply a *warp perspective* to correct any angle the image was taken from. In the end, the paper area is *cropped* with a default margin of -5 pixels. We have chosen this value as a minimum default to ensure that no background image will be inserted in the cropped area, while preserving most of the paper area. Note that this step will fail if the edges of the paper are not perfectly straight. In such cases, those margins should be changed separately for any direction.

3.1.3. Thresholding (Binarization)

In this step, the paper area image is binarized with adaptive thresholding of a 201 size Gaussian kernel window and a subtracted constant 13. Those values were found by trial-and-error and worked very well with every image we tried. However, they should be changed if the application produces unsatisfying binarization.

Note that we also compared other types of thresholding, such as classic and OTSU, which use global threshold values, but they yielded lower quality results. Additionally, thresholding inverts the image in a way so that it has white characters (pixel values = 1) in a black background (pixel values = 0). Figure 2 shows the comparison of the above thresholding algorithms.

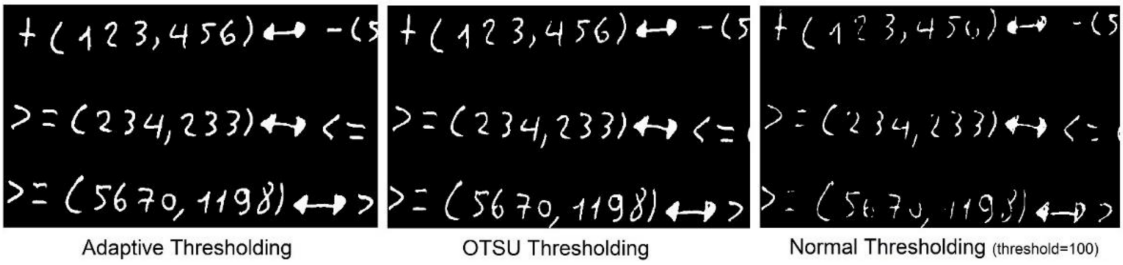


Figure 2. Comparison of thresholding algorithms.

As shown in Figure 2, normal thresholding with 100 fixed values produces poor results, while we lose much information from the characters. With the OTSU thresholding method, we had better results, but again, a little amount of information was lost. Adaptive thresholding was the best, and for more noise-free results, a median filter was applied first with a small-sized erosion after thresholding.

3.1.4. Detection of Text Area

To detect the text area on a thresholded image, we collect all points whose pixel values are greater than their average and find the *minimum rotated rectangle* that encloses them. Then, the rectangle's angle is computed and corrected with *warp affine transformation*. Finally, the corrected rectangle area is *cropped*. For better results, we apply hard *Gaussian blurring* at the start of this step to avoid any noise pixels that may lead to the wrong angle of the rectangle. Note that if there are any large noise pixels out of the text area, this step may fail and find a totally wrong rotated rectangle. The whole process is shown in Figure 3.

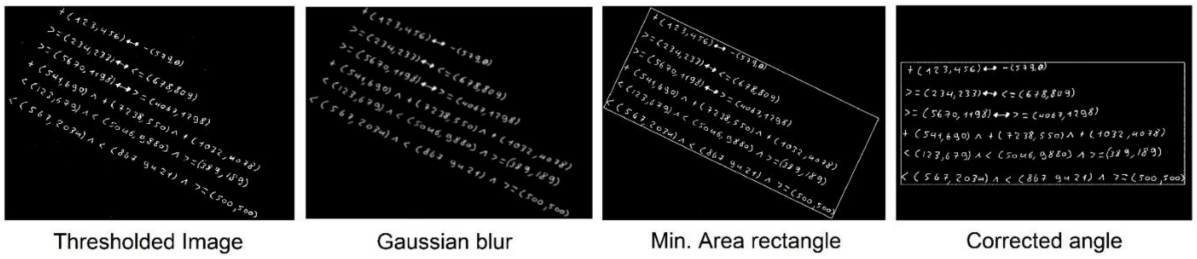


Figure 3. Example of text area detection.

In Figure 3, we start from the far-left binarized image, apply blurring with a 21 size Gaussian kernel window, find the enclosing minimum rotated rectangle from pixels that are greater than their average value, calculate the new angle, and warp affine the image to deskew it. In the end, for the cropping, we insert a padding and crop image with a +10 pixels additional margin, to avoid any extrapolated pixels.

3.1.5. Character Detection (Contours Extraction)

This is the most important step, as the method finds all possible contours in an image with the Suzuki algorithm [22] and stores them in a matrix variable (Mat class of OpenCV). Note that image resolution is crucial for better results of the algorithm. In addition, the assumption here is that no contour character should be linked to another or they will be considered as a single one. However, the problem of two-part characters like 'i', 'j' and '=' arises, which we solve in a further step. After every character is isolated, the method marks them with a red rectangle, as shown in the example of Figure 4.

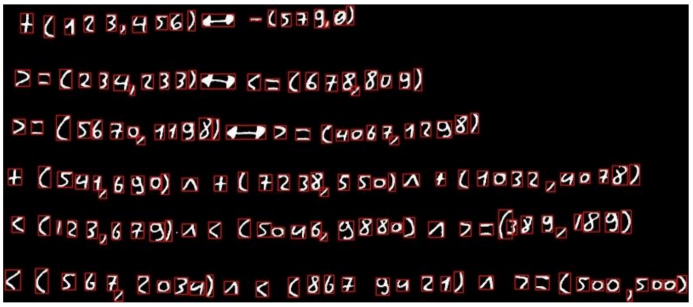


Figure 4. Example of contours extraction.

3.1.6. Line Detection

Detection of handwritten text lines is quite challenging because of the large variance in line angles and cross interferences, etc. We designed an algorithm based on profile projection, as described in [23,24]. Figure 5 shows an example of horizontal profile projection on printed and handwritten text.

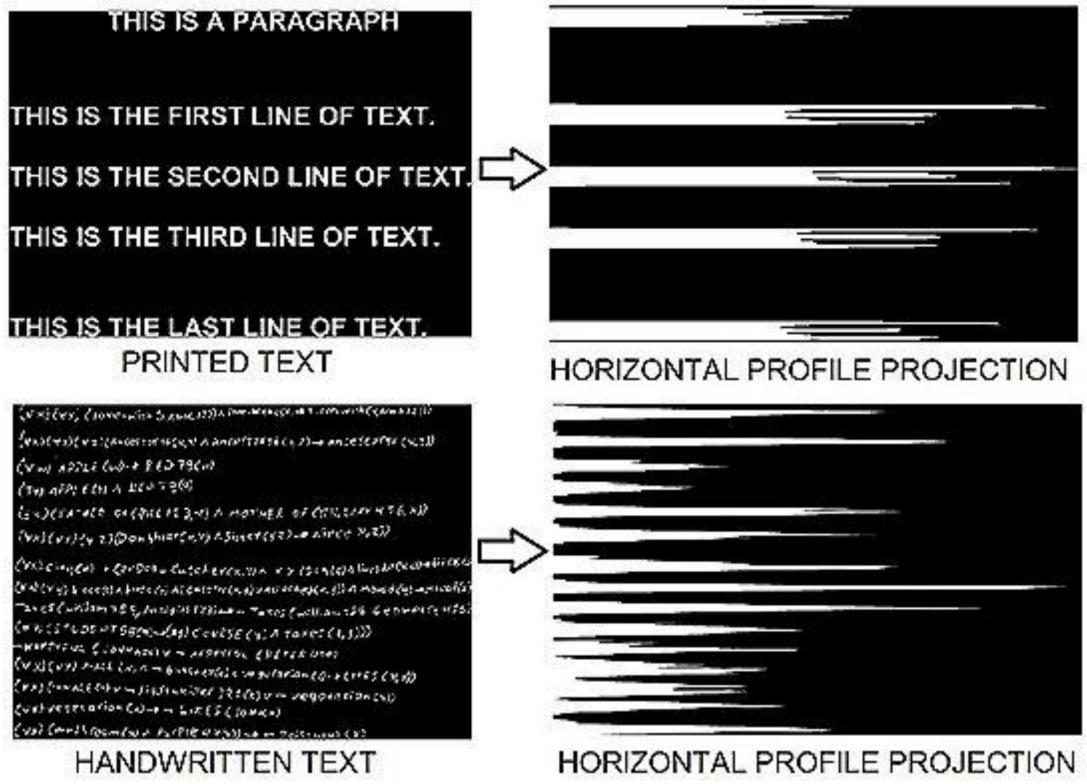


Figure 5. Horizontal profile projection examples.

As shown in Figure 5, horizontal profile projection creates a histogram by counting white pixel values horizontally. In the case of printed text, the distinction of lines is very clear, but in the case of handwritten text, we see that the bottom histogram peaks interfere with each other, which is a common problem of handwriting on blank paper. Therefore, to solve this problem, we propose the following algorithm:

1. Compute the average height and width of characters.
2. Create an image with filled rectangles of every character and decrease its' height by 2/3 if it is greater than the average height.
3. Divide the image in vertical stripes (partitions) of width equal to 3 times the average width.

4. Calculate horizontal histograms of every stripe and draw rectangles on lines' bound areas.
5. Concatenate all stripes and find all formed contours (as possible lines).
6. Search for small contours with height less than 2/3 of the average height, and width less than half of the whole image width and merge them with the closest contour.
7. Merge all contour pairs that have vertical distance less than average height (the case of lines that have large gaps inside).
8. Order contour lines by vertical position and find every character that lies inside each line based on its' center.
9. Calculate the convex hull of each line according to its' characters and draw the overlay (for user display).

This algorithm yielded 100% accuracy for all our image samples but is not perfect because it makes assumptions such as low variant character sizes, no connection between characters from consecutive lines, etc. Figure 6 shows an example of the algorithm's basic steps applied to a binarized image sample.

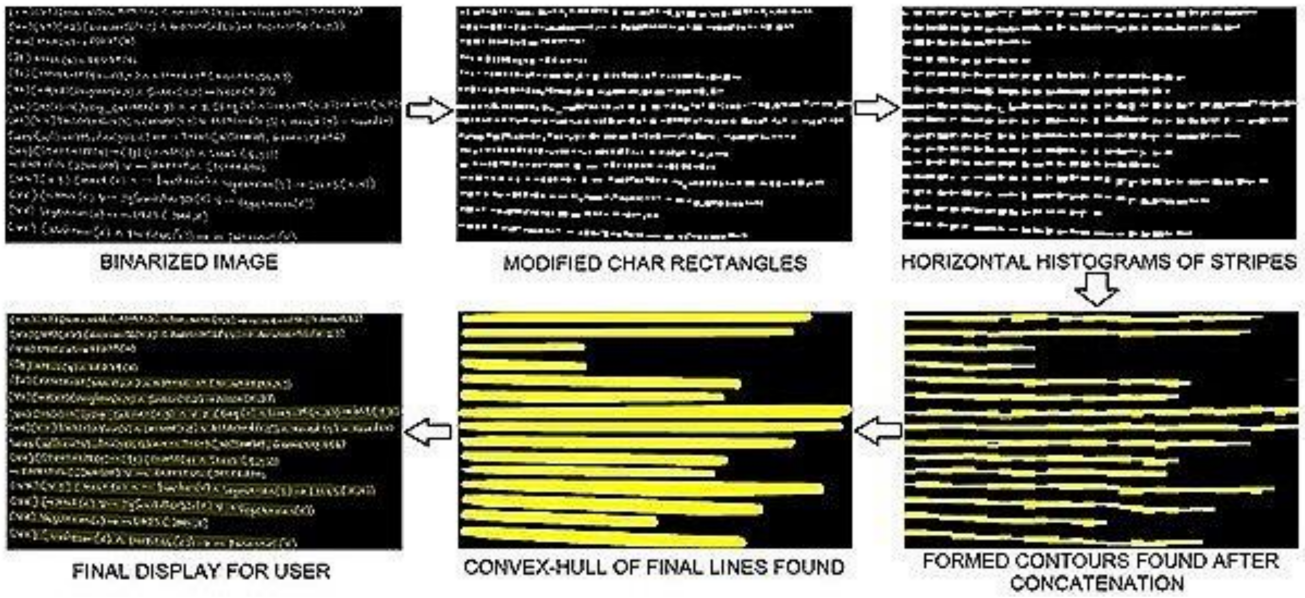


Figure 6. Basic steps of the line detection algorithm.

As shown in Figure 6, we start from the binarized and cropped text area, create an image of characters' modified rectangles, divide it into vertical stripes, find the horizontal histogram for every stripe, draw a rectangle at every possible line area and concatenate them all again forming the bottom-right image. Next, we find all contours and merge the smaller ones with those closest to them. Then we draw their convex-hull outlines (based on the characters of each line) and we transparently overlay them with the characters of the binarized image, as shown in the bottom-left image. Finally, the red marking rectangles of Figure 6 are drawn. This is performed just for user display reasons and is the last shown optical image of the image preprocessing stage.

3.1.7. Ordering and Normalizing Characters

After all of the lines are detected, we order every character found in each line by the *x*-axis, and we merge the two-part characters 'i', 'j' and '='. The idea behind merging is to take into account the geometric and position features of the three special characters separately. Therefore, we traverse all adjacent pairs of characters in every line and for:

- 'i' and 'j': check if the lower point of the small one is on top of the higher point of the big one. At the same time, check if the slope between them is greater than 1.

- '=': check if both aspect ratios are greater than 2 and if the center of one lies inside the other's width area.

After that, very small characters with an area less than 50 (this value was set after experimentation) are removed. With this method, we had 100% success on all our samples.

In this step, we also normalize every character to obtain the same size. The dimensions of every character found were squared, preserving its aspect ratio, and afterward were resized using the INTER-AREA interpolation method. This *resizing* method was selected because it showed better results with smoothed images and good quality. For testing purposes, we created two images of different size for each character: one of 28 × 28 and the other of 35 × 35 pixels. As a result, we tried two different trainsets—one corresponding to 28 × 28 size images and the second to 35 × 35 size images.

Additionally, we tried *thinning* every character with the Zhang–Suen algorithm [25] to test its impact on training. It seems that the Zhang–Suen algorithm works well in general, but a closer look reveals that some information is lost, which can lead to low recognition performance of the network.

In this final step, because of resizing, characters have pixel values in the range 0–255; therefore, we *re-binarize* them to obtain binary values again. Therefore, by traversing every matrix row on every normalized character, we store pixel values in a csv. file, separating them by commas and changing the line when the next character comes.

3.2. Neural Network Design and Training

In Figure 7, the main steps of this stage are depicted.

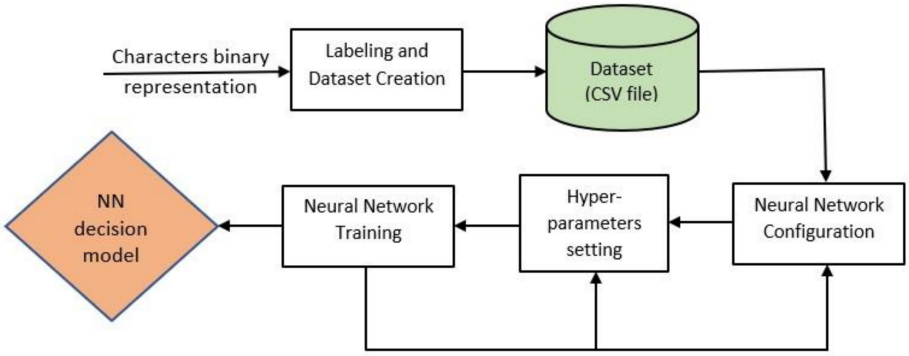


Figure 7. Neural network design and training flowchart.

3.2.1. Labeling and Dataset Creation

In labeling, binary representations of characters resulting from image processing are given labels. Each label is a representation of the corresponding character. We have chosen a one-hot representation for labels. Our vocabulary includes all English alphabet characters (lowercase and capital), all digits (0–9) and the special characters {"<", ">", "-", "+", "=", "(", ")", ",", "∀", "∃", "∧", "∨", "→", "↔", "¬"}, most of which are necessary for the representation of logic formulas. We must note that the characters that look identical (e.g., v-V-∨, c-C) are considered once and have a single representation. Finally, 67 unique characters resulted. Each one of them is represented as a vector with value '1' in the corresponding place, and '0' in all other places. We store the resulted binary representations in a CSV file. Therefore, the CSV file includes binary tuples of size 784 (=28 × 28) + 67 = 851 (in the case of 28 × 28 images) or 1225 (=35 × 35) + 67 = 1.292 (in the case of 35 × 35 images). This is the produced dataset.

3.2.2. Neural Network Configuration

After creating the dataset with the desired labels, we are ready to train the neural network (NN). To this end, we first define the architecture of the NN, i.e., define the:

- Number of inputs,

- Number of outputs,
- Number of hidden layers,
- Number of neurons at each hidden layer,
- Activation function of neurons at each layer.

The number of inputs is the same as the number of image pixels (784 or 1225), and the number of outputs is the same as the number of words in our vocabulary (67). The number of hidden layers and the neurons at each layer are set by trial-and-error. We are experimenting with many different cases and choose the best (see Section 4).

3.2.3. Hyperparameters Setting

There are various algorithms for NN training. Depending on the algorithm used, there may be different hyperparameters to be set. The following, except one, are common all the algorithms we used:

- Learning rate,
- Batch size,
- Maximum number of epochs for training,
- Momentum (only for the backpropagation algorithm).

We follow the same strategy here: they are set by trial-and-error.

3.2.4. Neural Network Training

Neural network training includes:

- Initialization of weights,
- Selection of training algorithm,
- Testing and re-training.

For initialization of weights, we use the algorithm proposed by Xavier Glorot et al. [26], which showed faster convergence and better performance in training. Furthermore, shuffling of trainsets resulted in better accuracy, so we applied this for every training experiment. A regularization technique, called Dropout, where a portion of neuron connections is cut in every epoch, along with adaptive learning rate and early stopping strategies, was used in some training to monitor whether final accuracy improves. By early stopping strategy, we mean that, if during a training, accuracy does not improve or decreases, then we stop.

We selected three training algorithms to test (implemented in the Encog 3.4 library), which belong to the Gradient Descent family. Their synoptic functions are as follows:

- Backpropagation: The basic technique that finds the error of the network by propagating backward and changes weights calculating the gradient and magnitude of the error function in every epoch. We can have incremental or batch training.
- Resilient Propagation: An improvement of backpropagation proposed by M. Riedmiller et al. [27], where the change of weights is calculated only by the gradient's sign and independently, avoiding the defining learning rate and momentum. It works well only with batch training.
- Stochastic Gradient Descent: It takes into account only a random portion (batch) of the trainset in each epoch and changes weights independently using the Adam update rule (proposed by Kingma and Ba [7]). This algorithm is also called mini-batch Gradient Descent.

To apply an algorithm, we need to somehow create a trainset from the produced dataset (for details, see Section 4) and a testset (the remaining cases). The application of an algorithm means running it for all cases in the trainset, according to the selected batch mode, many times (according to the number of epochs set), based on the current configuration and hyperparameter settings. Based on the results, the trial-and-error process may require strong recursion between this and the previous two steps (as shown in Figure 7). Even after achieving the best result, the application of the (candidate) model to the testset may lead to further training cycles. Finally, the best reached model is used as the final decision model.

4. Experimental Study

In this work, more than 100 long-term training experiments were conducted on different structures of networks, with the Softmax activation function in the output layer of 67 neurons and the shuffled trainset as commonly shared parameters. For every experiment, a graph plot of trainset–testset accuracy was created, including the error rate per epoch (recalibrated in the 0–100 interval). Because of randomized initial weights, we used the best-of-3 evaluation strategy, by which every experiment was conducted 3 times and the best result was recorded. All experiments were conducted on a laptop with 64-bit Windows, CPU Intel Core i5-8250U and 8GB RAM.

The evaluation of every trained network was performed by the following main criteria:

- Overall accuracy of network for testset,
- Accuracy per character and ability to distinguish look-alike characters by examining the Confusion Matrix of the testset,
- Impact of 28×28 versus 35×35 normalization on final accuracy,
- Generalizing ability of network (low variance between trainset–testset accuracy),
- Training time.

4.1. Formulation of Datasets

Initially, we selected 36 First Order Predicate Logic sentences with varying degrees of difficulty and a typed prototype was created (three pages of 15, 15 and 6 typed sentences, respectively, in landscape mode). Next, we gave the prototype to ten different people and collected three equivalent handwritten pages from each (30 pages, 360 sentences in total). Then, the handwritten pages were scanned with 400 dpi resolution in the TIF lossless format. The initial number of total isolated characters was 16,822, but after examining their occurrence frequency, we saw that they had a quite large imbalance leading to an unbalanced dataset (trainset).

We expected that all frequencies in samples would be 10 times the prototype frequencies (because of 10 copies of the prototype). However, results showed some differences, due to the following: some people made mistakes by ignoring or miswriting characters, so in labeling, we had to change corresponding initial labels to have a more realistic matching (e.g., some persons in writing 't' missed its bottom part, so it looked like '+' and we labeled it as that).

To deal with this imbalance, we used the techniques of under-sampling and image data augmentation, where characters with more than 250 occurrences were removed and those with less than 250 were rotated in an angle range of -17 to +17 to create additional distorted versions of them. Therefore, the final trainset size was 16,750 with 250 samples per character. The threshold of 250 was selected as a good balance value on average. A smaller threshold would lose too much of the original samples, while a bigger one would lead to too many distorted samples. Furthermore, note that we considered the distribution per person seriously, so we created distorted versions using original characters from every person's writing style. As for the testset, we retained all the removed samples from the under-sampling technique (occurrences that exceeded the 250 threshold) and created an additional smaller handwritten dataset (four extra pages) that was merged with them. The extra dataset was created because some characters did not have any removed samples. The final testset had 7947 samples.

4.2. Training with Shuffled–Unshuffled Trainset

After having created the trainset from the formulated dataset, we had to determine if shuffling should be used or not in our experiments. In Figure 8, we present four accuracy graphs of our first experiments with unshuffled and shuffled versions of trainset using the backpropagation training algorithm and sigmoid activation function, which show the impact of shuffling on a network's training and accuracy.

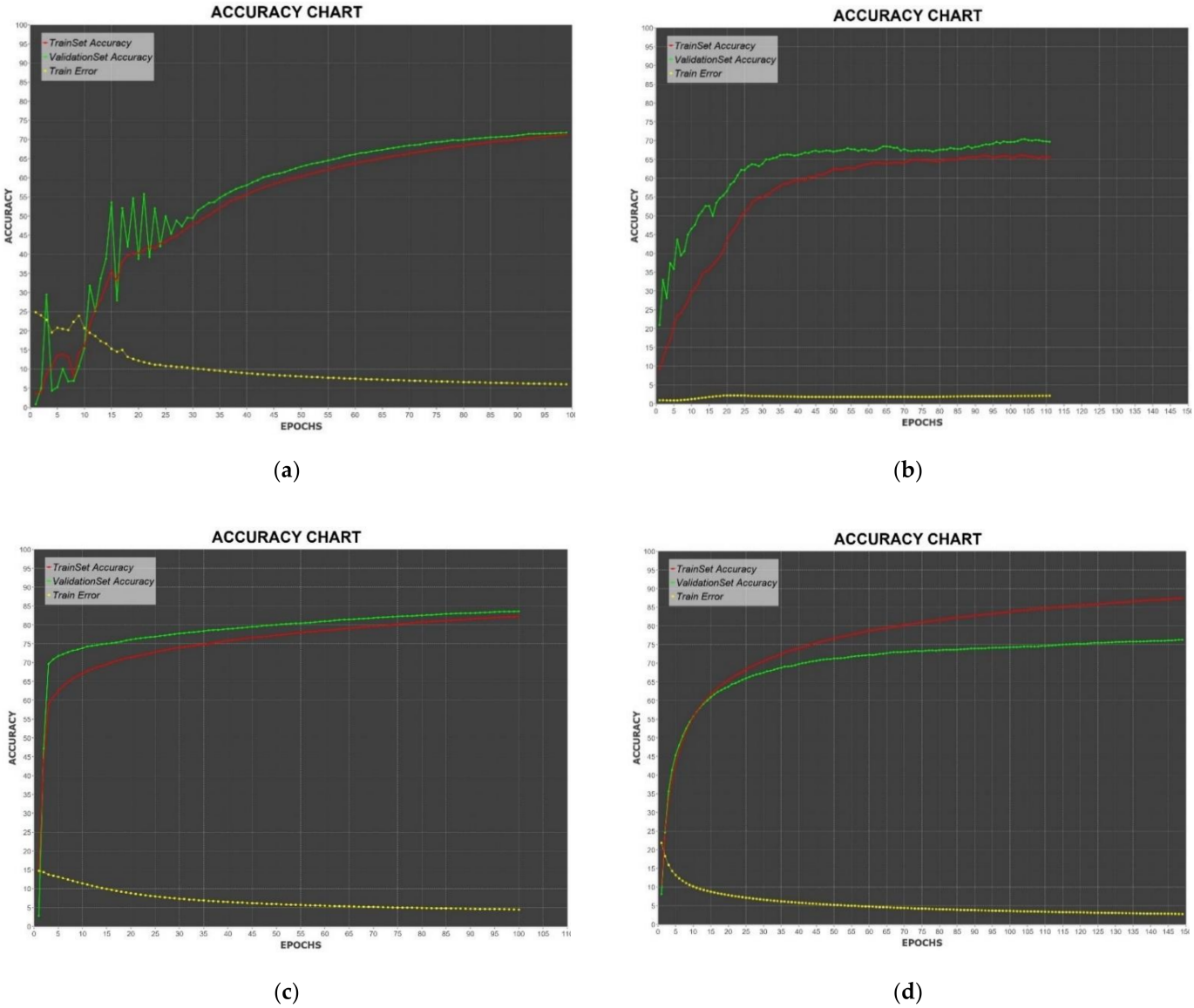


Figure 8. Impact of shuffled–unshuffled trainset: (**a**) unshuffled trainset, backpropagation, full batch; (**b**) unshuffled trainset, backpropagation, batch = 1; (**c**) shuffled trainset, backpropagation, full batch; (**d**) shuffled trainset, backpropagation, batch = 1.

As shown in Figure 8, accuracy and smoothness are higher with a shuffled trainset than with an unshuffled one. More precisely, in the case of an unshuffled trainset, there is great instability during training, while in the case of shuffled, there is a stable, smooth accuracy curve. Additionally, we must note that shuffling is necessary when we use incremental or online training, while it is meaningless in a full batch, as all trainset samples are used in order to accumulate weight changes and they are applied at the end of every epoch.

4.3. Training with Different Layer Sizes

A wide set of parameters and layers were examined to define the best structure of hidden layers. We trained different networks with different layer sizes using all three algorithms, learning rate of 10^{-4} and different activation functions, such as sigmoid, Elliott [28] and rectified linear unit (ReLU). The performances of the most indicative networks are presented in Table 1.

Table 1. Training with different hidden layer sizes and activation functions.

Training Algorithm	Hidden Layer Size	Batch Size	Activation Function	Testset Accuracy
Backpropagation	200	1	Sigmoid	82.06%
RPROP	200	Full	Elliott	59.05%
SGD	200	512	ReLU	85.1%
Backpropagation	300	1	Elliott	83.2%
RPROP	300	Full	Sigmoid	48.7%
SGD	300	512	ReLU	90%
Backpropagation	400	1	Elliott	82.8%
RPROP	400	Full	Sigmoid	55.17%
SGD	400	512	ReLU	90.02%
Backpropagation	500	1	Sigmoid	76.2%
RPROP	500	Full	Elliott	65.8%
SGD	500	512	ReLU	88.98%
Backpropagation	700	1	Sigmoid	73.1%
RPROP	700	Full	Sigmoid	74.9%
SGD	700	512	ReLU	89.7%
Backpropagation	800	1	Elliott	88.5%
RPROP	800	Full	Elliott	62.25%
SGD	800	512	ReLU	90.03%
Backpropagation	900	1	Elliott	87.9%
RPROP	900	Full	Sigmoid	75.7%
SGD	900	512	ReLU	90.1%

In Table 1, we can see that, depending on the training algorithm, different accuracies are achieved. In backpropagation, accuracy increases inconsistently, achieving its best (88.5%) with 800 neurons at the hidden layer. After size 800, accuracy drops again. Furthermore, the training times were very long, reaching even 4–5 h for sizes above 500. Note that in all cases, backpropagation was used with incremental training (batch = 1).

In RPROP, we see a quite large increase in accuracy from 300 to 700 neurons and slightly better performance with 900 neurons, where the highest accuracy reached 75.7%. Because of its nature, RPROP worked well only with full batch training, and training times were better than those for backpropagation.

In SGD, we had an 85.1% accuracy starting even with the small size of 200 neurons and, above 300–400, did not manage to significantly improve the 90% limit. Its' training times were the best among all three algorithms, and the batch sizes were powers of 2.

For a better comparison of the three algorithms, we additionally present in Figure 9 an overall graph of testset accuracies depending on hidden layer sizes.

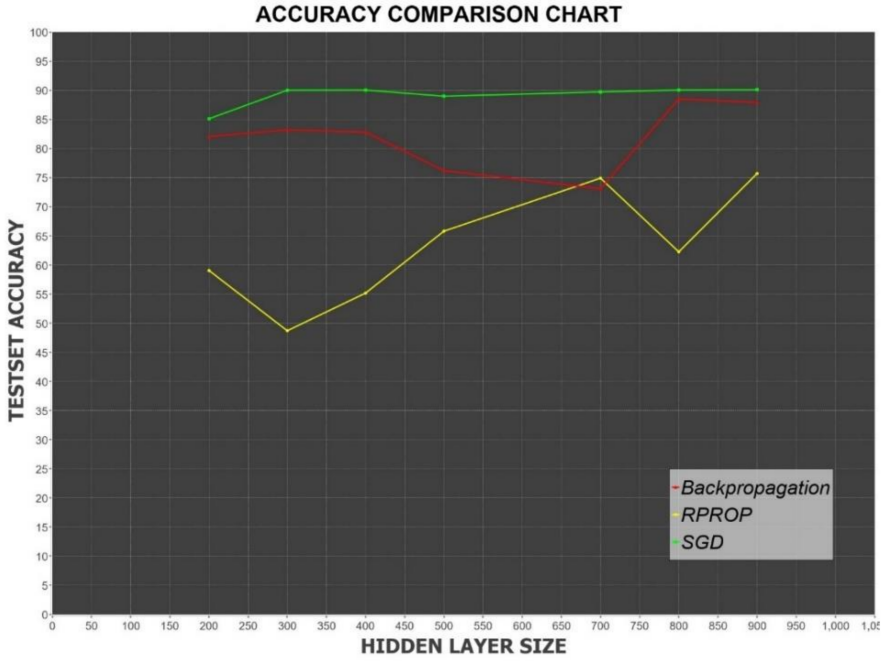


Figure 9. Comparison of backpropagation-RPROP-SGD testset accuracies of Table 1.

From the corresponding accuracy graphs (not shown here), it shows that in the SGD algorithm, the trainset's accuracy had a significant improvement as the hidden layer size was increasing, while testset accuracy did not have that much. However, its fluctuation increased too. In backpropagation graphs, a smoother accuracy is shown with less variance between the trainset and testset as the layer size was increasing. Finally, RPROP had quite unstable accuracies in both the trainset and testset, with some serious drops after the 50th epoch. This drop effect happened in almost every RPROP experiment we tried, so the accuracies presented here were the highest achieved during overall training.

After experimenting with one layer of various sizes, we trained some networks with two hidden layers, as shown in Table 2.

Table 2. Training NNs with 2 hidden layers.

Training Algorithm	Hidden Layer 1	Hidden Layer 2	Activation Function	Testset Accuracy
Backpropagation	300	300	Elliott	87.6%
Backpropagation	400	400	Elliott	88.8%
SGD	400	400	Elliott	89.4%
RPROP	300	300	Sigmoid	62.6%

From Table 2 and the accuracy graphs (not shown here), we see that trainset accuracy reached 100% for the backpropagation algorithm in both different sizes of 300-300 and 400-400, but testset accuracy remained stable. This means that training led to overfitting. SGD showed almost similar results but with a little lower variance and a fluctuating testset accuracy. On the other hand, RPROP had much lower accuracies with a huge drop at the end. Furthermore, we experienced the strange phenomenon of a significantly greater testset accuracy than trainset one during all training. It is obvious that by adding extra hidden layers, the network's complexity increases.

4.4. Training with Different Activation Functions

Among various available activation functions, we selected three: logistic sigmoid, Elliott and ReLU. Tanh and linear functions proved inappropriate for our problem. In the

following training graphs in Figure 10, the impact of them using backpropagation in a network with one hidden layer of 400 neurons and 50 epochs maximum training is shown.

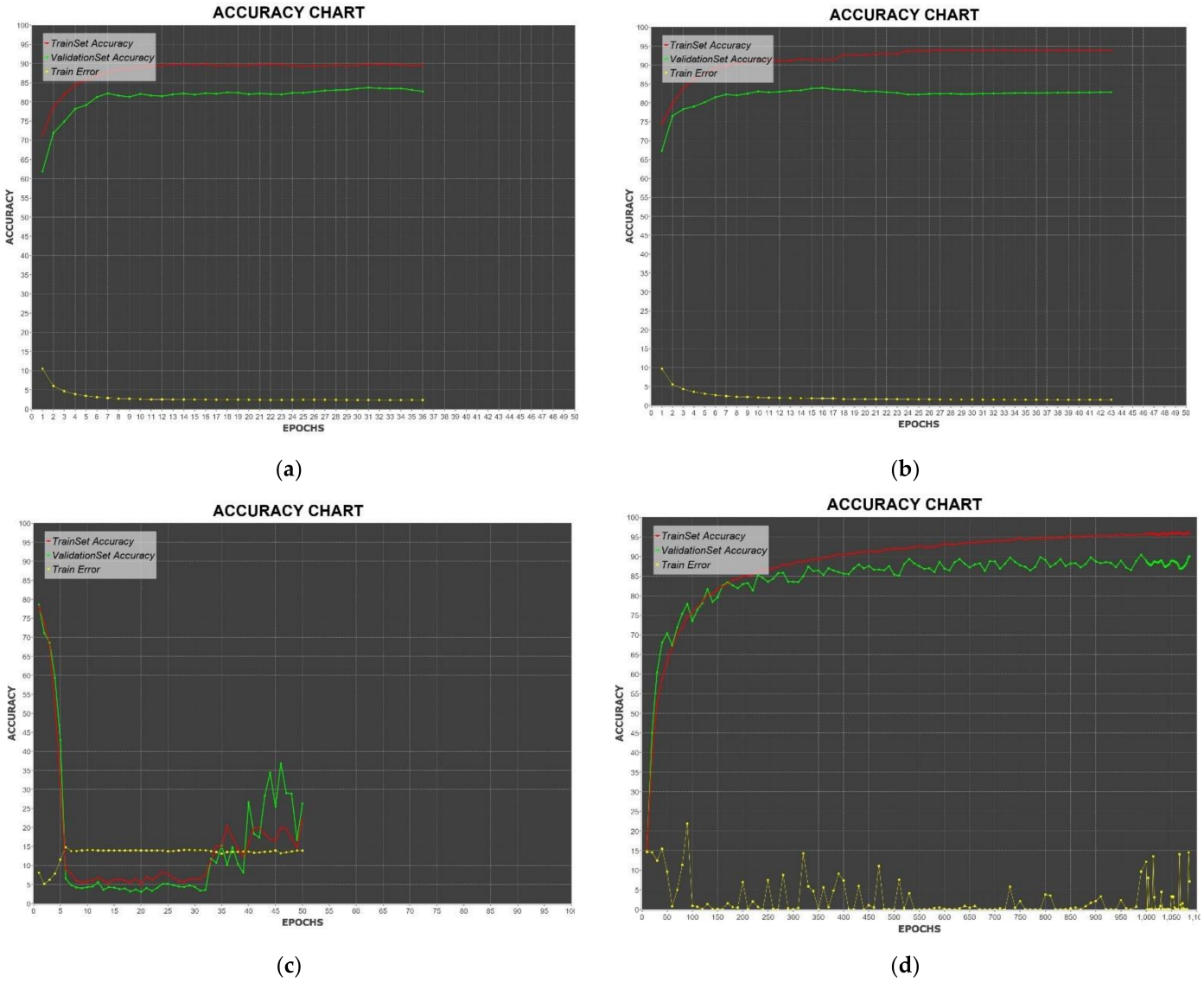


Figure 10. Comparison of activation functions. (**a**) Training with the logistic sigmoid function; (**b**) training with the Elliott function; (**c**) training with the ReLU function (backpropagation); (**d**) training with the ReLU function (SGD).

As we see, sigmoid and Elliott functions led to a quite smooth accuracy movement, with Elliott slightly improving it. On the other hand, the ReLU function had a terrible performance with backpropagation with a steep error increase, but with SGD, it achieved the best accuracy compared to all three functions.

4.5. Training with Dropout Technique and Adaptive Learning Rate

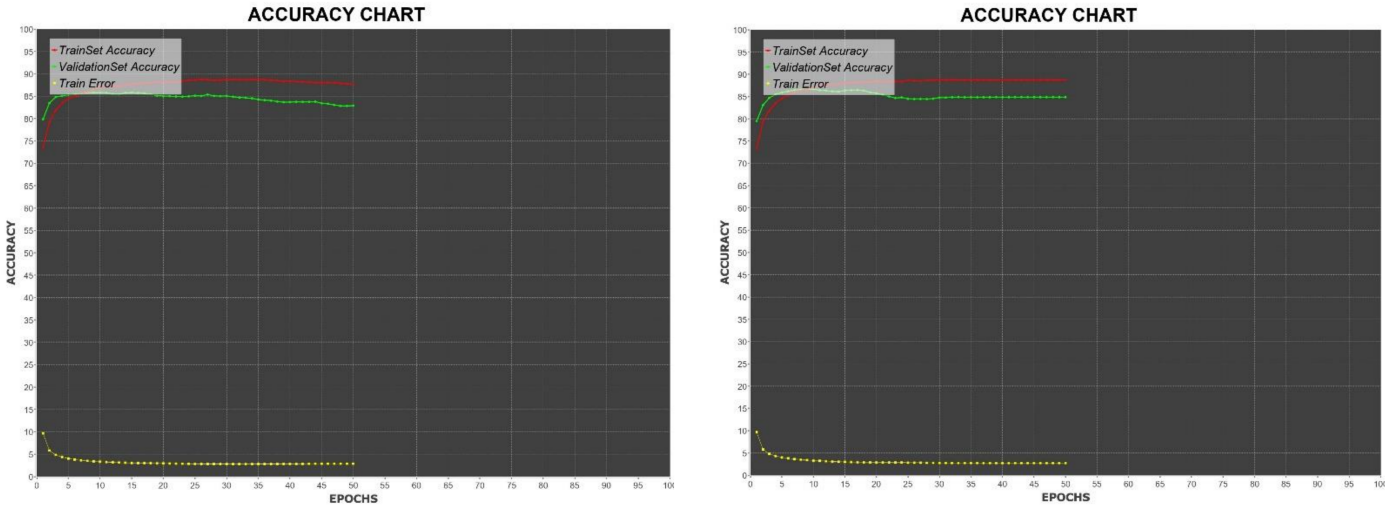
In some promising trainings, we used the Dropout regularization technique to see if testset accuracy improves. With this technique, a number of randomly selected neuron outputs are ignored in every epoch in order to avoid some neurons to co-adapt and make the network more sensitive to their weights. In Table 3, we show the effect of Dropout in all three cases. The decimal number represents the percentage of randomly selected neuron connections that were ignored in every epoch.

Table 3. Training with the Dropout technique.

Training Algorithm	Hidden Layers	Activation Function	Trainset Accuracy	Testset Accuracy
Backpropagation	400-400	Elliott	100%	89%
Backprop. (Dropout: 0.5)	400-400	Elliott	96.7%	89.1%
Backpropagation	800	Elliott	90%	88.5%
Backprop. (Dropout: 0.8)	800	Elliott	88.6%	88.9%
RPROP	500	Elliott	80.3%	80.9%
RPROP (Dropout: 0.1)	500	Elliott	77.6%	77.4%

In the first case of backpropagation with a Dropout of 0.5, we see an insignificant increase in testset accuracy (by 0.1%), while the trainset drop is 3.3%. In the second case, with a Dropout of 0.8, we had a slightly bigger increase in the testset, and also a decrease in the trainset. In the RPROP case, with a minimum Dropout of 0.1, we had a decrease in both trainset and testset accuracies. In general, the testset accuracy did not improve much, but the variance of the network was decreased.

In Figure 11, we display the training graphs before and after applying the adaptive learning rate strategy, in which the learning rate is decreased by half, every time the epoch error increases. For this experiment, we used the backpropagation algorithm with one hidden layer of 400 neurons, the Elliott activation function, momentum with a value of 0.9 and initial learning rate equal to 0.0005.



(**a**) (**b**)

Figure 11. Comparison of fixed and adaptive learning rate training: (**a**) training with fixed learning rate 0.0005; (**b**) training with adaptive learning rate.

As we see from Figure 11, a fixed learning rate led to a decaying accuracy, while the adaptive learning rate kept it stable. This is a reasonable expectation, as during training, the network tries to minimize the error function. When it reaches close to a global minimum, the learning rate (or the step) should be changed in order to adapt the sensitivity of the network changes to weights.

4.6. Training with Different Batch Sizes

The best training algorithm to show the effect of the batch size is SGD, where we used sizes of 256, 512, 1024 and the whole trainset size (full batch) in a network with one hidden layer of 400 neurons, a ReLU activation function and learning rate of 0.001. The

corresponding training graphs along with their confusion matrices and a table (Table 4) with the highest accuracies achieved for each are presented below.

Table 4. Testset accuracies on different batch sizes.

Training Algorithm	Hidden Layer Size	Batch Size	Testset Accuracy
SGD	400	256	88.77%
SGD	400	512	90.13%
SGD	400	1024	89.7%
SGD	400	full	88.39%

By looking at the accuracy graphs (not shown here), we clearly see the greater fluctuation of testset accuracy while the batch size is decreasing. In the case of a full batch, we see that both error and accuracy move more smoothly, and the graph could be similar to a backpropagation training. Table 4 shows that the best accuracy is achieved with a batch size of 512, while with smaller or greater sizes, the accuracy has a little drop.

4.7. Training with Normalization Sizes 28 × 28 and 35 × 35

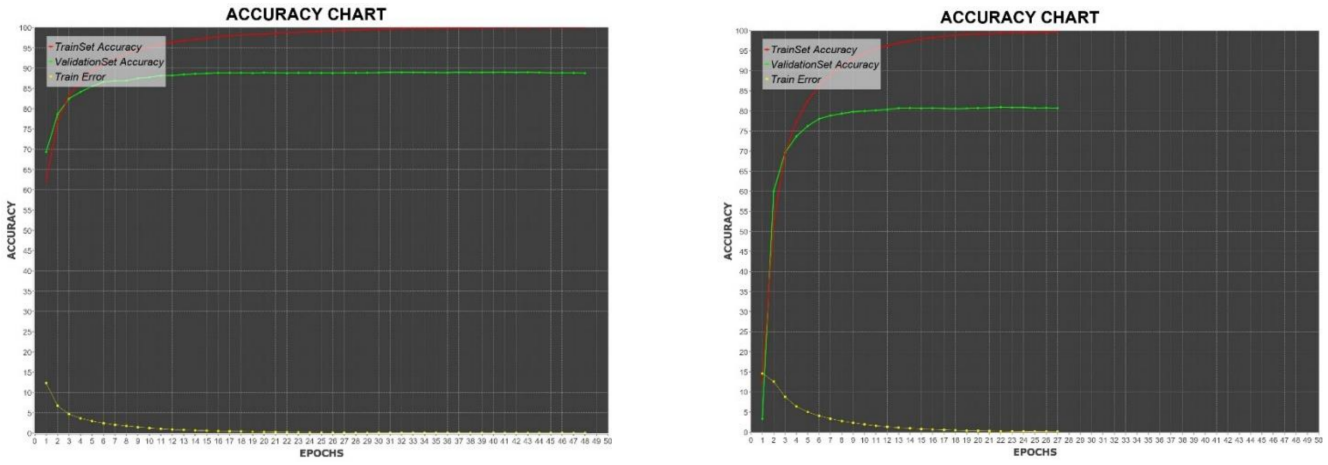
In the previously presented results, most networks were trained with the 35 × 35 trainset and the rest with the 28 × 28 one. Table 5 displays results for both cases for backpropagation and SGD algorithms.

Table 5. Accuracy of the trainset–testset on 28 × 28 and 35 × 35 normalization.

Training Algorithm	Hidden Layer Size	Batch Size	Activation Function	Trainset Accuracy	Testset Accuracy
Backprop. (28 × 28)	800	1	Elliott	92.6%	87.3%
Backprop. (35 × 35)	800	1	Elliott	92.48%	87.21%
SGD (28 × 28)	400	512	ReLU	92.19%	90.13%
SGD (35 × 35)	400	512	ReLU	94.24%	90.17%

From Table 5, we see that testset accuracy did not have any significant difference in both normalization sizes and algorithms. However, in SGD with 28 × 28 normalization, there is a smaller variance in the trainset–testset than that with 35 × 35.

As an additional experiment, we tried *thinning* every character with the Zhang–Suen algorithm to test whether the final accuracy improved. The following training graphs (Figure 12) show the thinning impact using a 35 × 35 trainset and backpropagation with two hidden layers of 400 neurons, the Elliott activation function, learning rate of 0.0005 and momentum of 0.9.



(**a**) (**b**)

Figure 12. Training comparison with normal and thinned characters. (**a**) Training with normal characters (35 × 35); (**b**) training with thinned characters (35 × 35).

Here, we clearly see much better performance with normal characters (higher testset accuracy and smaller variance) than with thinned (lower testset accuracy and much bigger variance).

Finally, we must mention that, as shown in some previous graphs, the epochs of training were less when we did not see any improvement in order to save time (early stopping strategy). The SGD algorithm needed about 1000 epochs in most experiments, whereas backpropagation and RPROP need around 50–100. In SGD cases, we were calculating accuracy every 10 epochs because it was an extra time-consuming process.

5. Application Development

An application was developed that implements our methodology and is able to recognize handwritten logic formulas. The application was developed in Java and performs all the steps that were presented in the methodology section. Its operation is schematically displayed in Figure 13.

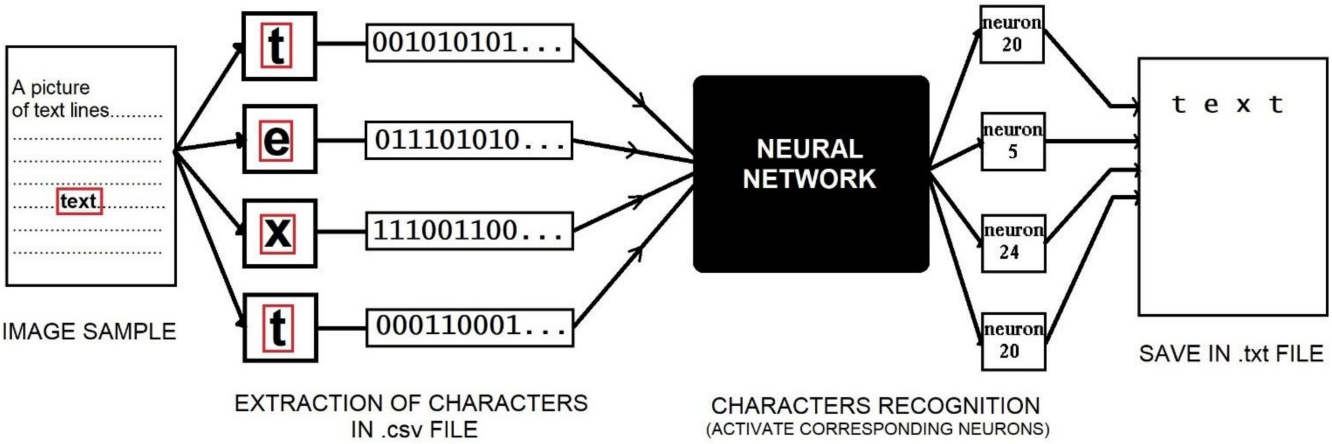


Figure 13. Basic schematic operation of the application.

5.1. Basic Functionalities

The application's graphical interface (see Figure 14) lets the user select an image file of JPEG or TIFF format. Then, the image is preprocessed, and the characters are extracted and saved in a csv. file. During this phase, basic steps of image processing are shown to the

user with a 1 s time delay interval between them (see Figure 15, for instance). Next, the pretrained NN is loaded, and the csv. file is given as input. For each line read in the file (every character extracted), we specify the output neuron with a higher value closer to 1 and decode it to the corresponding character. At the same time, a txt. file is created and, considering the detected line structure, every recognized character is written into it.

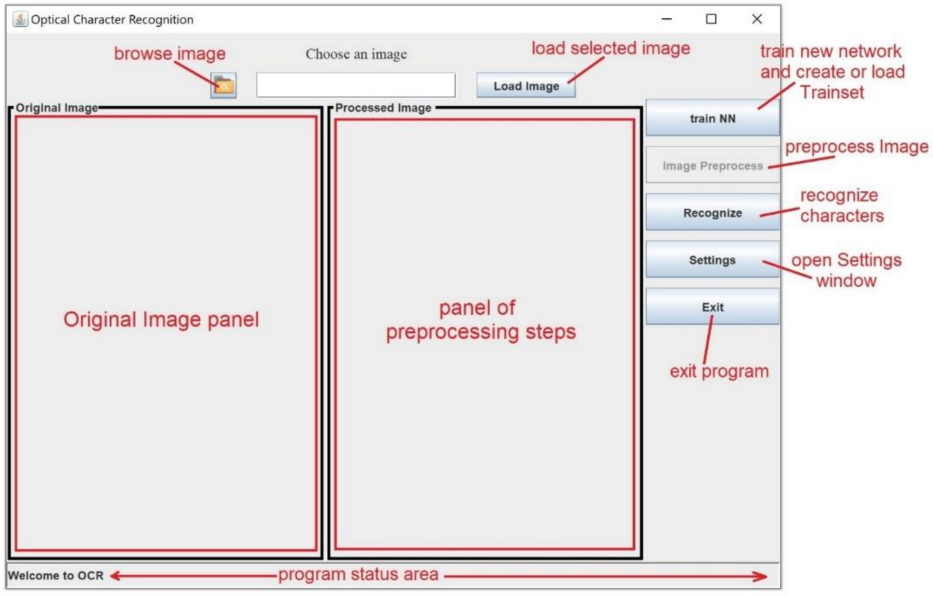


Figure 14. Main interface window of the application.

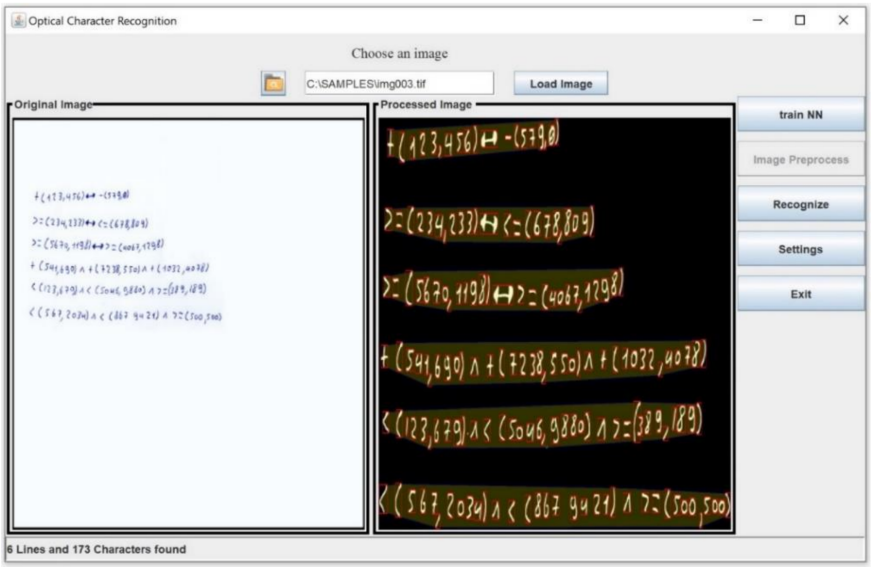


Figure 15. Main window instance of the final preprocessing stage.

5.2. Additional Functionalities

An additional function of our application is the creation of a new trainset. The user can input an image of his/her preferred character samples, preprocess it as previously described, and click the 'Train NN' button. To this end, a new window is shown, where the user can give the desired labels to the newly extracted character samples. They are validated by the system and saved in a csv. file. In the case of an invalid label, the process recurses; the csv. file can act as the new trainset.

Another additional function is the capability to create and train a new user-defined NN model. The parameters of the network are initially set to the application's default

values, but the user can change them by pressing the 'Settings' button to move to the settings window, where the user can redefine the number of neurons of the hidden layer, the activation functions, the max epochs, the learning rate and the batch size.

6. Discussion of Results

From the presented results, we can see the high complexity of training a neural network due to the tuning requirements of various parameters. In this section, we present the findings that came out of our experimentation.

Trainset size is one of the most important factors in HCR accuracy. Today's modern commercial HCR applications that use NNs as classifiers are trained with huge datasets of samples with a large variety per class (character) and achieve recognition accuracies above 99%. Furthermore, shuffling a trainset can make quite a big difference in final accuracy (Figure 8). Note that this makes sense only for incremental or mini-batch training. Another important factor is the image quality. For example, the lighting conditions of a taken picture, its resolution, the surrounding context of text in an image, etc., affect the creation of a good model.

Focusing on NNs, the random initialization of weights can affect training, by causing longer training times, extremely small values (vanishing gradient problem) or even no convergence. The Xavier initialization method we used showed much better performance in general and should always be used to assure faster training and convergence.

6.1. Effect of Hidden Layers

By examining the results of Tables 1 and 2, we make the following observations:

- Increasing the size of a hidden layer can lead to better accuracy till a certain point. Of course, by increasing the size, the network complexity is also increased, meaning that it can approximate more complex functions, but after a point, this will lead to memorizing the trainset without improvement of the testset (overfitting).
- By adding a second hidden layer, the network complexity increases even more with the trainset reaching 100% accuracy faster in some algorithms. Furthermore, testset accuracy may be improved a little, but generally, this results in greater variance of trainset–testset. Thus, the network learns the trainset perfectly but has low generalizing ability on new datasets.

According to the above, we understand that more than one hidden layer is redundant for this particular trainset used and that the hidden layer size should be increased until testset accuracy has no significant improvement. On the other hand, using smaller sizes has the advantage of less computational power needed for training but increases the risk of underfitting (lower accuracy on both the trainset and testset). A neuron size range of 400–800 seems to be adequate for our problem.

6.2. Effect of Activation Functions

The comparison of logistic, Elliott, and ReLU activation functions, as shown in Figure 10, underlines the following effects:

- Sigmoid and Elliott behaved quite similarly in the backpropagation algorithm, in contrast to ReLU, which showed unpredictable and very low accuracy. Between sigmoid and Elliott, we believe Elliott is better for two reasons: its plot has a smoother curve, which leads to better classification detail, and it needs less computational power, thus lower training times.
- ReLU performed much better with SGD, achieving the best accuracy overall. We assume that the bad performance in backpropagation was caused by the exploding gradient problem that may have happened because of ReLU's steep linear part, leading to very big weight values and accumulation of errors.
- The tanh function proved to be inappropriate for our network because of its output range from -1 to 1, while we encoded from 0 to 1.

- We believe that the Softmax function was the best choice for the output layer, as it is ideal for letting only one active neuron out, based on the possibilities (one-hot encoding).

6.3. Effect of Dropout and Learning Rate

In Table 3, we see that by applying the Dropout regularization technique, the variance of the trainset–testset decreased, but the testset accuracy had no significant improvement. Namely, it was like trying to slow down the learning process by forcing the network to deploy all of its neurons' connections but with much longer training time. Although the decrease in variance made the network more capable of generalizing, the testset accuracy did not have any improvement.

Regarding the learning rate, we see that it can mostly affect training time and sometimes accuracy. If a large value is defined, then it leads to faster learning initially, but with unstable and fluctuating errors (or decreasing accuracy). If a small value is defined, then we have a slower learning curve and run the risk of becoming stuck at a local minimum of error function. As shown in Figure 11, our implementation of an adaptive learning rate strategy seems to stabilize the accuracy but does not improve it. However, in the SGD algorithm, the use of the Adam optimization method also had the effect of an adaptive strategy, where we defined only the initial learning rate and then the weights changed in a way that improved the results (compared to the other two algorithm accuracies achieved). The Adam method calculates adaptive learning rates for every weight independently, based on the first and second 'moment' of gradients. Therefore, we understand that adaptive learning rate strategies proved quite useful for training, considering Adam as the one with the best results for our problem.

6.4. Effect of Batch Size

Considering the four batch sizes we used for SGD, we observed that smaller sizes result in rapid learning but with volatile and fluctuating accuracy stats. On the other hand, larger batch sizes showed slightly slower learning but with more stabilized accuracy. This is expected if we examine how SGD works with mini-batches. If a small portion of the trainset is taken into account in every epoch, then the network estimates the average error from that portion. Hence, comparing the error of all samples and the error of a part of them, there will always be some inconsistency leading to larger deviations of accuracy between epochs. As shown in Table 4, in the case of full batch size, the accuracy was smoother and reached 88.39%, while with smaller sizes, greater accuracies were achieved even instantly with better training times.

Nevertheless, corresponding confusion matrices seem quite similar, meaning that the network performed almost the same on character distinction ability with minor differences. Furthermore, note that batch size values should be powers of 2. In the backpropagation algorithm, different batch sizes resulted only in better training times but not in higher accuracy.

We believe that the SGD algorithm should not use very large batch sizes because it loses its stochastic nature. Smaller sizes are better to use because, despite the higher fluctuation, we can instantly achieve higher testset accuracy and shorter training times.

6.5. Effect of Normalization (Resizing-Thinning)

Considering the results of Table 5 and the particular trainset used, we did not see any significant difference in accuracy. In the SGD case, the testset accuracy improved slightly with 35×35 normalization, but if we examine the equivalent confusion matrices, only two characters had low classification in the 28×28 trainset ('k' and '∃' under 50% accuracy), and four characters in the 35×35 trainset ('k', '0', '9' and '∃' under 50% accuracy). Therefore, resizing characters to dimensions of 28×28 led to better generalization of the network, as confirmed by Table 5, where variance is smaller than the 35×35 case (92.19–90.13% < 94.24–90.17%).

Considering an intuitive view, we can think about the amount of detail between the two dimensions. With the 35 × 35 normalization, there is more detail on samples, but the network will have to be more complex and have a larger margin of errors. On the other hand, 28 × 28 normalization has less detail, but the network will be simpler with fewer errors and of course a much better training time.

Regarding the thinning experiment with Zhang–Suen algorithm, as shown in Figure 12, the network had a lower accuracy and a bigger variance. Although we thought that thinning would improve the recognition, this did not happen, and we believe that it is due to the Zhang–Suen algorithm (as shown in the example of Figure 7, where some samples were distorted) and to the reduction of white pixels that made the network less sensitive to any character variations.

6.6. Choice of Training Algorithm

As we compare all three algorithms in all presented results, we make the following observations:

- RPROP had good training times, but it resulted in the worst accuracy with unstable and sometimes unexplained performance. For example, we noticed a significant drop in accuracy after the 70th epoch, while the testset accuracy was higher than the trainset accuracy during all of the training. Although Encog's developer suggests using RPROP over backpropagation, as it is more sophisticated, we assumed that it was inappropriate for our kind of problem. The highest accuracy achieved with this algorithm was 80.9%.
- Backpropagation produced quite good results with ideal parameters: Elliott activation function, adaptive learning rate with an initial value of 0.0005, and a momentum of 0.99. However, the largest problem was long training times (an average of 2–4 h per experiment) with 50–100 max epochs. In all of the presented training graphs, we see a large accuracy from the first epoch (greater than 60%), followed by slow improvements. This happened because of the initial learning rates and proper activation functions used (Logistic and Elliott). The highest accuracy we achieved was 89.1%, with two hidden layers of 400 neurons.
- The SGD algorithm achieved the best accuracies and, by using the ReLU activation function and proper batch size, managed to pass the 90% accuracy limit. Furthermore, it had the shortest training times, smaller trainset–testset variance and slightly better confusion matrices compared to backpropagation. We believe that its success is due to its stochastic nature and the Adam optimization method implemented in the Encog library. The highest accuracy achieved was 90.13%, and that is why, by default, we consider it as the best choice for the final trained network in our application.

6.7. The Final Network of Application

According to all previous remarks, we defined the default pretrained network that is used by our application in order to perform HCR tasks. The final parameters that are defined and used in the application are the following:

- Character Normalization: 28 × 28,
- Input Layer: 784 inputs,
- Hidden layers: one layer of 400 neurons with ReLU activation function,
- Output layer: 67 neurons with Softmax activation function,
- Training algorithm: stochastic gradient descent with Adam optimization,
- Initial learning rate: 0.001,
- Batch size: 512.

This neural network achieved a final testset accuracy of 90.13% and trainset accuracy of 92.19% with an early stopping strategy. In total, 42 of 67 characters were recognized with an accuracy greater than 90%, while characters 'k' and '∃' with less than 50%. Therefore, this network has a low ability to distinguish 'k' from 'x' and '∃' from '3'. Finally, we must

note that the accuracy of 90.13% depends on this particular testset used and does not imply that it will be the same for every other testset.

7. Comparisons

In Table 6, we attempt to compare our work with other recent works that use NN for English handwritten character recognition based on basic parameters ('?' means "estimated" because it is not clear in the corresponding paper).

Table 6. Comparison of approaches for handwritten character recognition.

Research Work	Dataset (Samples)	Character Type	NN Architecture	Training Algorithm	Accuracy
[8] Perwej and Chaturvedi (2011)	650 (520 train, 130 test), size 5 × 5	English lowercase alphabet	2 hidden layers (25-25-25-?)	Backpropagation (?)	82.5%
[10] Choudhary etal (2013)	1300, size 15 × 12	English lowercase alphabet	1 hidden layer (180-80-26)	Backpropagation, adaptive learning rate	85.62%
[11] Katiyar and Mehfuz (2015)	CEDAR: 21,328 (19,145 train, 2183 test)	English alphabet	2 hidden layers (144-100-90-6) hybrid features	Backpropagation	93.23%
[13] Attigeri (2018)	4840 (4400 train, 440 test), size 30 × 20	English lowercase alphabet	two hidden layers of 100 neurons	Backpropagation	90.19%
[16] Yousaf et al. (2019)	HCD: 27,142 (19422 train, 7720 test), size 60 × 40	English capital alphabet Digits	1 hidden layer: 2400-240-26 (alphabet) 2400-120-10 (digits)	Backpropagation (?)	96.98% (alphabet) 98.08% (digits)
[17] Kosykh et al. (2019)	MNIST: 70,000 (60,000 train, 10,000 test), size 28 × 28	Digits	1 hidden layer (784-145-10)	SGD	95.7%
Our Approach	24,697 (16,750 train, 7947 test), size 28 × 28	English alphabet Digits Logic characters	1 hidden layer (784-400-67)	SGD with Adam optimization	90.13%

Based on the information from Table 6, we can conclude that our approach has reasons to be characterized as among the best, if not as the best. The approach in [11], although it declares slightly more than 2% better accuracy, it lacks the ability to recognize logic characters, which has added an extra level of difficulty to our approach. Furthermore, a more time-consuming feature extraction approach is used. The approach in [13] had an easier task to tackle: recognition of only lowercase letters. The approach in [16], although it declares 7+% better accuracy, it is a result of much easier recognition tasks: recognizing only capital letters or only digits. Finally, the approach in [17], although it shows about 5.5% better performance, it is also due to the much easier task of recognizing only digits and a much larger data set was used. On the other hand, none of the works in Table 6 report any systematic experimental study related to hyperparameters.

8. Conclusions

In this paper, we present an HCR system that aims to recognize first-order logic handwritten formulas and create corresponding editable text. We introduce a methodology for creating a suitable data set from images of the handwritten formulas and producing a trained feedforward dense neural network decision model. The methodology includes two main stages: image processing and NN design and training. In the first stage, we produce a dataset by using image processing techniques on images of handwritten logic formulas. The dataset consists of binary tuples representing handwritten characters. In the second stage, NN architecture is designed and trained with the produced dataset in a recursive way due to the required tuning of its parameters and hyperparameters.

We have executed a large number of experiments to find the best configuration of the trained NN, in terms of character normalization size, number of inputs, number of output neurons, number of hidden layers and their neurons, activation functions, training

algorithms, learning rate and batch size. The best accuracy achieved was 90.13%. This accuracy is very comparable with the results of other similar approaches, if not the best.

From the experiments, we extracted many useful remarks concerning various sides of the training algorithms in relation to the parameters and hyperparameters of the NN. Stochastic gradient descent with Adam optimization turned out to be the best training algorithm for our problem. Furthermore, ReLU and Softmax were proved to be the best activation functions for the hidden layer and the output neurons, respectively.

Finally, we presented a system developed in Java, which implements the above methodology and creates a decision model. Additionally, it gives the user the possibility to re-train the NN, i.e., to produce a different model, by creating new datasets based on new data and re-configure the NN architecture and training process by redefining the parameters and hyperparameters of the NN.

The results of this work are based on a specific dataset, so their validity is not fully assured. The use of several datasets and the production of new results is a direction for further work. On the other hand, the use of a deep learning neural network architecture, such as CNN, could be another direction of future work, where a much bigger data set will be needed.

Author Contributions: Conceptualization, I.P., I.H. and V.A.; methodology, V.A., I.P. and I.H.; software, V.A.; validation, I.P. and I.H.; formal analysis, V.A. and I.P.; investigation, V.A.; writing—original draft preparation, V.A. and I.P.; writing—review and editing, I.H. and G.T.; supervision, I.H. and G.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original datasets of handwritten first-order predicate logic sentences characters along with their extra 400 rotated versions (per character) can be found at https://github.com/scisor/OCR-FOPL_Datasets (accessed on 17 October 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rajalakshmi, M.; Saranya, P.; Shanmugavadivu, P. Pattern Recognition-Recognition of Handwritten Document Using Convolutional Neural Networks. In Proceedings of the 2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS), Tamilnadu, India, 11–13 April 2019; pp. 1–7. [CrossRef]
2. Gao, X.; Deng, H.; Ma, M.; Guan, Q.; Sun, Q.; Si, W.; Zhong, X. Removing light interference to improve character recognition rate by using single-pixel imaging. *Opt. Lasers Eng.* **2021**, *140*, 106517. [CrossRef]
3. Jiao, S.; Feng, J.; Gao, Y.; Lei, T.; Yuan, X. Visual cryptography in single-pixel imaging. *Opt. Express* **2020**, *28*, 7301–7313. [CrossRef] [PubMed]
4. Vinjit, B.M.; Bhojak, M.K.; Kumar, S.; Chalak, G. A Review on Handwritten Character Recognition Methods and Techniques. In Proceedings of the 2020 International Conference on Communication and Signal Processing (ICCSP), Melmaruvathur, India, 28–30 July 2020; pp. 1224–1228. [CrossRef]
5. Ahlawat, S.; Choudhary, A.; Nayyar, A.; Singh, S.; Yoon, B. Improved Handwritten Digit Recognition Using Convolutional Neural Networks (CNN). *Sensors* **2020**, *20*, 3344. [CrossRef] [PubMed]
6. El-Sawy, A.; Loey, M.; El-Bakry, H. Arabic Handwritten Characters Recognition using Convolutional Neural Network. *WSEAS Trans. Comput. Res.* **2017**, *5*, 11–19.
7. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference for Learning Representations, San Diego, CA, USA, 7–9 May 2015.
8. Perwej, Y.; Chaturvedi, A. Neural Networks for Handwritten English Alphabet Recognition. *Int. J. Comput. Appl.* **2011**, *20*, 2449–2824. [CrossRef]
9. Kader, F.; Kaushik, D. Neural Network-Based English Alphanumeric Character Recognition. *Int. J. Comput. Sci. Eng. Appl. (IJCSEA)* **2012**, *2*, 1–10. [CrossRef]
10. Choudhary, A.; Rishi, R.; Ahlawat, S. Off-Line Handwritten Character Recognition using Features Extracted from Binarization Technique. *AASRI Procedia* **2013**, *4*, 306–312. [CrossRef]
11. Katiyar, G.; Mehfuz, S. MLPNN based handwritten character recognition using combined feature extraction. In Proceedings of the International Conference on Computing, Communication & Automation, Greater Noida, India, 15–16 May 2015; pp. 1155–1159. [CrossRef]

12. Afroge, S.; Ahmed, B.; Mahmud, F. Optical character recognition using back propagation neural network. In Proceedings of the 2016 2nd International Conference on Electrical, Computer & Telecommunication Engineering (ICECTE), Rajshahi, Bangladesh, 8–10 December 2016; pp. 1–4. [CrossRef]
13. Attigeri, S. Neural network based handwritten character recognition system. *Int. J. Eng. Comput. Sci.* **2018**, *7*, 23761–23768. [CrossRef]
14. Chen, A.W. Handwriting Recognition and Prediction Using Stochastic Logistic Regression. *Int. J. Inf. Res. Rev.* **2018**, *5*, 5526–5527.
15. Jana, R.; Bhattacharyya, S.; Das, S. Handwritten Digit Recognition Using Convolutional Neural Networks. In *Deep Learning: Research and Applications*; Bhattacharyya, S., Snasel, V., Hassanien, A.E., Saha, S., Tripathy, B.K., Eds.; De Gruyter: Berlin, Boston, 2020; pp. 51–68.
16. Yousaf, A.; Khan, M.J.; Khan, M.J.; Javed, N.; Ibrahim, H.; Khursid, K. Size Invariant Handwritten Character Recognition using Single Layer Feedforward Backpropagation Neural Networks. In Proceedings of the 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), Sindh, Pakistan, 30–31 January 2019; pp. 1–7. [CrossRef]
17. Kosykh, N.E.; Khomonenko, A.D.; Bochkov, A.P.; Kikot, A.V. Integration of Big Data Processing Tools and Neural Networks for Image Classification. In Proceedings of the Models and Methods of Information Systems Research Workshop 2019 (MMISR 2019), St. Petersburg, Russian, 4–5 December 2019; pp. 52–58.
18. Parthiban, R.; Ezhilarasi, R.; Saravanan, D. Optical Character Recognition for English Handwritten Text Using Recurrent Neural Network. In Proceedings of the 2020 International Conference on System, Computation, Automation and Networking (ICSCAN), Puducherry, India, 27–28 March 2020; pp. 1–5. [CrossRef]
19. Bora, M.B.; Daimary, D.; Amitab, K.; Kandar, D. Handwritten Character Recognition from Images using CNN-ECOC. *Procedia Comput. Sci.* **2020**, *167*, 2403–2409. [CrossRef]
20. Ahlawat, S.; Choudhary, A. Hybrid CNN-SVM Classifier for Handwritten Digit Recognition. *Procedia Comput. Sci.* **2020**, *167*, 2554–2560. [CrossRef]
21. Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Trans. Syst. Man Cybern.* **1979**, *9*, 62–66. [CrossRef]
22. Suzuki, S.; Be, K.A. Topological structural analysis of digitized binary images by border following. *Comput. Vis. Graph. Image Process.* **1985**, *30*, 32–46. [CrossRef]
23. Ha, J.; Haralick, R.M.; Philips, T.I. Document page decomposition by the bounding-box project. In Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, Canada, 4–16 August 1995.
24. Kunte, S.R.; Sudhaker, S.R.D. A simple and efficient optical character recognition system for basic symbols in printed Kannada text. *Sadhana* **2007**, *32*, 521–533. [CrossRef]
25. Zhang, T.Y.; Suen, C.Y. A Fast Parallel Algorithm for Thinning Digital Patterns. *Commun. ACM* **1984**, *27*, 236–239. [CrossRef]
26. Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, Sardinia, Italy, 13–15 May 2010; pp. 249–256.
27. Riedmiller, M.; Braun, H. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In Proceedings of the IEEE International Conference on Neural Networks, San Francisco, CA, USA, 28 March–1 April 1993; pp. 586–591.
28. Elliott, D.L. *A Better Activation Function for Artificial Neural Networks*; Technical Report 93-8; Institute for Systems Research, University of Maryland: College Park, MD, USA, 1993.